Universidad Autónoma de Madrid

Facultad de Ciencias
Máster en Matemáticas y Aplicaciones

TRABAJO DE FIN DE MASTER

Dependent Typing through Closed Cartesian Categories

Presentado por:
Pedro Bonilla Nadal

Dirigido por:
Ángel González Prieto

Curso académico 2020-2021

Dependent Typing through Closed Cartesian Categories

Pedro Bonilla Nadal

Pedro Bonilla Nadal *Dependent Typing through Closed Cartesian Categories*

.

Master's thesis.
Academic year 2020-2021.

Thesis supervisor

Dr. Ángel González Prieto
Departamento de Matemáticas
Universidad Autónoma de Madrid

Máster en Matemáticas y Aplicaciones

Universidad Autónoma de Madrid

Contents

Part I

Category Theory

Chapter 1

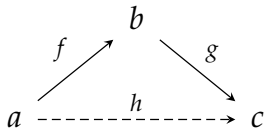
First Notions of Category Theory

> "The human mind has never invented a labor-saving machine equal to algebra."
>
> *Stephan Banach (1925)*

For us Category Theory is an area of interest on its own rights, rather than merely an elegant tool. Thus, we will introduce this theory with a point of view that emphasize the intuitive ideas behind each notion.

With this objective in mind, we will get into the habit of introducing the first examples even before introducing the formal definitions. Thus, when the formal definition is introduced it becomes meaningful.

The fundamental idea of Category Theory is that many properties can be unified if expressed in arrow diagrams. Intuitively, a diagram is a directed graph, such that each way of going from a node to another are equals. For example, the diagram:



Means that $f \circ g = h$. This approach to mathematics emphasises at the relationships between elements, rather than at the structure of the elements themselves. In general, dashed lines means that the existence of that particular arrow is uniquely determined by the solid arrow presents in the diagram.

In this first chapter we introduce the notion of category and some useful properties. The principal references for this chapter are [19] and [21]. Haskell references can be checked in [20].

1.1 Metacategories

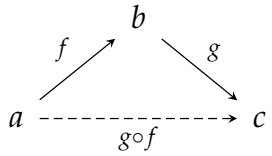
We will start by defining a concept independent of the set theory axioms: the concept of *metacategory*. Then, categories will arise from studying these concepts within set theory. We follow [19] for these definitions.

Traditionally, mathematics is based on the set theory. When we start set theory it is not necessary (it is not possible) to define what a set is. It is similar with the concepts of element and belonging, which are basic to set theory. Category theory can also be used to found mathematics. This theory provide meaning based on other concepts such as object, arrow or composition rather than sets or belonging.

Definition 1.1.1. A *metagraph* consist of *objects*: $a, b, c..$ and *arrows* $f, g, h....$ There are also two pairings: dom and codom. This pairings assigns each arrow with an object. An arrow f with $\mathrm{dom}(f) = a$ and $\mathrm{codom}(f) = b$ is usually denoted as $f : a \to b$.

Definition 1.1.2. A metacategory is a metagraph with two additional operations and two properties::

- Operations:
  - *Identity*: assigns to each object a an arrow $1_a : a \to a$.
  - *Composition*: assigns to each pair of arrows f, g with $\mathrm{codom}(f) = \mathrm{dom}(g)$ and arrow $g \circ f$ such that the diagram:



    commutes. The arrow $g \circ f$ is called the *composite* of f and g.

- Properties:
  - *Associative*: given arrows f, g, h, we have that,

$$(g \circ f) \circ h = g \circ (f \circ h).$$

  - *Unit*: given an object a, and arrows f, g such that $\mathrm{dom}(f) = a$ and $(g) = a$, we have that,

$$1_a \circ g = g, \qquad f \circ 1_a = f.$$

In the context of categories and metacategories, arrows are often called *morphisms*.

We have just define what a metacategory is without any need of set and elements. In most cases we will rely in a set theory interpretation of this definitions, as most examples (specially with our interest in computational example) will rely on this theory. Nonetheless, whenever possible, we will define the concepts working only in terms of objects and arrows.

1.2 Set theory categories

Despite being an alternative foundation of mathematics, in general we will work with categories interpreted with notions of set theory. Unless otherwise stated, we will use the Zermelo-Fraenkel axiomatics with Axiom of Choice. We recommend [17] for more information.

We have no interest in this work in the reformulation of axioms. Note that we have defined a meta-category without the need for notions of sets. In the same way, many definitions and propositions relating to this theory can be considered without the need for notions of sets.

Further on we will work based on set theory. Despite that, it is useful to take into account that most of the concept here presented can be defined without making use of this Theory.

Definition 1.2.1. A category (resp. graph) is an interpretation of a metacategory (resp. metagraph) within set theory.

That is, a graph/category is a pair (O, A) where a collection O consisting of all objects as well as a collection A consisting of all arrows. Their elements hold the same properties that objects and arrows hold on metacategories / metagraph.

We will focus on the category. First, we define the function homeset of a category $C = (O, A)$, wrote as $\hom_C$, as the function:

$$\begin{aligned} \hom_C : O \times O &\mapsto \mathcal{P}(A) \\ (a, b) &\mapsto \{f \in A | f : a \to b\} \end{aligned}$$

When there is no possibility of confusion we will state $c \in C$ without specifying . We will refer to the collection of objects of a category C as $Ob(C)$ and the collection of arrows as $Ar(C)$.

Definition 1.2.2. We say that a category is *small* if the collection of objects is given by a set (instead of a proper class). We say that a category is *locally small* if every homesets is a set.

We proceed to introduce a comprehensive list of examples, so that it is already introduced in subsequent chapters.

Example 1.2.1.

- The elementary categories:
  - The category $0 = (\emptyset, \emptyset)$ where every property is trivially satisfied.
  - The category $1 = (\{e\}, \{1_e\})$.
  - The category $2 = (\{a, b\}, \{1_a, 1_b, f : a \to b\})$
- Discrete categories: are categories where every arrow is an identity arrow. This are sets regarded as categories, in the following sense: every discrete category $C = (A, \{1_a : a \in A\})$ is fully identified by its set of object.
- Monoids and Groups: A monoid is a category with one object (regarding the monoid of the arrows). In the same way, if we requires the arrows to satisfy the inverse property, we can see a group as a single-object category.
- Preorder: From a preorder $(A, \leq)$ we can define a category $C = (A, B)$ where B has an arrow $e : a \to b$ for every $a, b \in A$ such that $a \leq B$. The identity arrow is the arrow that arise from the reflexive property of the preorders.
- Large categories: these categories has a large set of objects. For example:
  - The category *Top* that has as objects all small topological spaces and as arrow continuous mappings.
  - The category *Set* that has as objects all small sets and as arrows all functions between them. We can also consider the category Set_* of pointed small sets (sets with a distinguish point), and function between them that maps one distinguish point into another.
  - The category *Vect* That has as object all small vector spaces and as arrows all linear functions.
  - The category *Grp* That has as object all small vector group and as arrows all homomorphism.
  - The category Top_* that has as object all small topological spaces with a distinguished point, and as arrow all continuous functions that maps each distinguished points into distinguished points. Similarly we can consider Set_* or Grp_*.
- The category *Hask* of all Haskell types and all possible functions between two types.

Note that, for example, as natural numbers can be seen as either a set or a preorder, they also can be seen as a discrete category or a preorder category.

Simple enough categories can be easily described with diagrams: there are as many objects as nodes in the diagram and an arrow in the category for each:

- Arrow in the diagram.

- Composition of existing arrows.
- Every node (identity arrow usually omitted in diagrams).

For example, we can fully represent the 2 category with the diagram:

$$a \xrightarrow{\quad g \quad} c$$

1.2.1 Properties

We can see that is common in mathematics to have an object of study (propositional logic clauses, groups, Banach spaces or types in Haskell). Once the purpose of studying these particular sets of objects is fixed, it is also common to proceed to consider the transformations between these objects (partial truth assignments, homomorphisms, linear bounded functionals or functions in Haskell).

In categories, we have a kind of different approach to the subject. Instead of focusing on the objects themselves, we focus on how they relate to each other. That is, we focus on the study of the arrows and how they composes. Therefore we can consider equal two objects that has the same relations with other objects. This inspire the next definition:

Definition 1.2.3. [21, Definition 1.1.9] Given a category $C = (O, A)$, a morphism $f : a \to b \in A$ has a *left inverse* (resp. *right inverse*) if there exists a $g : b \to a \in A$ such that $g \circ f = 1_b$ (resp. $f \circ g = 1_a$). A morpishm is an *isomorphism* if it has both left and right inverse, called the *inverse*. Two object are isomorphic is there exists an isomorphism between them.

Is easy to follow that if a morphism has a left and a right inverse, they must be the same, thus implying the uniqueness of the inverse. Also one can see that this functions define bijections between $\hom(a, c)$ and $hom(b, c)$ for all $c \in O$.

We will now proceed to talk about certain arrows and objects that have properties that distinguish them from others. The most useful example to gain an intuition of these properties is that of the *FinSet* category, of all finite sets and functions between them. Considering special arrows:

Definition 1.2.4. An arrow f is *monic* (resp. *epic*) if it is *left-cancelable* (resp. *right-cancelable*), i.e. $f \circ g = f \circ h \implies g = h$ (resp. $g \circ f = h \circ f \implies g = h$).

In FinSet this arrows are the injective functions (resp. surjective). Considering special objects:

Definition 1.2.5. an object a is *terminal* (resp. *initial*) if for every object b there exists an unique arrow $f : b \to a$ (resp. $f : a \to b$). An object that is both terminal and initial is called *zero*.

In FinSet this objects is the initial object is the empty set and the terminal object the one point set. In the Pointed

Proposition 1.2.1. Every two terminal object are isomorphic.

Proof. Every terminal object has only one arrow from itself to itself, and necessarily this arrow has to be the identity. Let a, b be terminal object and $f : a \to b$ and $g : b \to a$ be the only arrows with that domain and codomain. Then $f \circ g : a \to a \implies f \circ g = 1_a$. Analogously $g \circ f = 1_b$.

□

Another important property is the *duality property*. This property tell us that for every theorem that we prove for categories, there exist another theorem that is automatically also true. To formalize this idea, we define the concept of *opposite category*.

Definition 1.2.6. [21, Definition 1.2.1] Let C be any category. The opposite category C^{op} has:

- the same objects as C,
- an arrow $f^{op} \in C^{op}$ for each arrow $f \in C$, so that the domain of f^{op} is defined as the codomain of f and viceversa.

The remaining structure of the category C^{op} is given as follows:

- For each object a, the arrow 1_a^{op} serves as its identity in C^{op}.
- We observe that f^{op} and g^{op} are composable when f and g are, and define $f^{op} \circ g^{op} = (g \circ f)^{op}$

The intuition is that we have the same category, only that all arrows are turned around. We can see that for each theorem T that we prove, we have reinterpretation that theorem to the opposite category. Intuitively this theorem is an equal theorem in which all the arrows have been turned around. For example: the proposition 1.2.1 can be reworked as:

Proposition 1.2.2. Every two initial object are isomorphic.

That is because being initial is the dual property of being terminal, that is, if $f \in C$ is a terminal object then $f^{op} \in C^{op}$ is an initial object. The property "being isomorphic" is its own dual.

1.2.2 Transformation in categories

This is one of the main ways of defining a category: consider a collection of objects and the standard way of transforms one into each other. Then, we may also follow our study defining the structure preserving transformation of categories.

Definition 1.2.7. Given two categories C, B, a *functor* $F : B \to C$ is a pair of functions $F = (F' : Ob(C) \to Ob(B), F'' : Ar(C) \to Ar(B))$ (the *object function* and the *arrow functor* respectively) in such a way that:

$$F''(1c) = 1_{F'c},\ \forall c \in Ob(C); \qquad F''(f \circ g) = F''f \circ F''g,\ \forall f, g \in Ar(C).$$

That is, a functor is a morphism of categories. When there is no ambiguity we will represent both F' and F'' with a single symbol F acting on both objects and arrows. Also, it can be seen in the definition that whenever possible the parentheses of the functor will be dropped. This loss of parentheses will be replicated thorough the text, whenever possible.

An important consideration on functors is that a functor F can be defined only pointing out how it maps arrows (assuming that such mapping respects the composition), as how F maps object can be defined with how it map identity arrows.

Lets provide some examples of functors:

Example 1.2.2.

- Forgetfull functor: We have a variety of categories consisting of structures with sets as objects and functions that hold structure such as arrows (eg Top, Grp or Vect).

  Let C one of such categories, then we have a functor $F : C \to Set$ that maps each object to its underlying set and each arrow to the equivalent arrow between sets. That is, this functor forgets about the structure that is present in C.

  Additionally, we will often have functors defined $F : C \to Set$. For some cases of C it also happen that FC has some more structure on it. In that case we will say that F is an enriched functor for that cases.

- Fundamental Group: In the context of algebraic topology we have the functor of the fundamental group $\Pi_1 : Top_* \to Grp$. The most famous property of this function is that

  > Given continuous application $f, g : X \to Y$ between two topological spaces induces an application of the group of loops of X on the group of loops of Y such that $\Pi_1(f \circ g) = \Pi_1(f) \circ \Pi_1(g)$.

  that is, the functoriality of the function (taking also into account the fact that the identity of topological spaces is mapped to the identity of group).

- Stone-Čech Compactification: In the realm of functional análisis we have the following theorem:

  Theorem 1.2.1. *Let Ω be a completely regular Hausdorff topological space. Then there exists a compact Hausdorff topological space $\beta\Omega$ and a homeomorphism Φ, from Ω to a dense subset of $\beta\Omega$. Moreover, for every continuous and bounded function $f : \Omega \to \mathbb{K}$ there exists a unique $\overline{f} \in C(\beta\Omega)$ such that $\overline{f} \circ \Phi = f$ and the application $f \to \overline{f}$ is an isometric isomorphism. Finally, $\beta\Omega$ is uniquely determined up to isomorphisms by the above properties.*

  Remark 1.2.1. Where $\Delta(t) = \delta_t$ denotes Dirac's operator.

  Proof. It is enough to take as $\beta\Omega$ the closure of $\Delta(\Omega)$ in the induced weak-topology ω^* (since in the weak topology we have that a bounded closed is compact).

  Given a continuous and bounded function $f : \Omega \to K$ we can define the function $\overline{f}$ by injection of Ω and then limit to maintain continuity (which will exist by compactness and boundedness of f). Thus the maximum of $\overline{f}$ can be reached by a sequence of points injected from Ω obtaining that

  $$\max\left\{ |\ \overline{f}\ (s)\ | : s \in \beta\Omega \right\} = \sup\{|f(s)| : s \in \Omega\}.$$

  The application $f \to \overline{f}$ is therefore an isometric isomorphism. Thanks to this isometric isomorphism the function spaces and by the Banach-Stone theorem[2, Theorem 3] we have uniqueness up to isomorphisms. □

  Based on this theorem we can define a functor $\beta : Hauss \to Comp$ where *hauss* is the category of Completely regular Haussdorff Spaces and continuous functions, and *Comp* the category of Compact Haussdorff Spaces along with continuous function. This functor:

  - Take an object $\Omega \in Hauss$ to $\beta(\Omega) = \beta\Omega$:
  - Take an arrow $f : \Omega \to \Omega' \in Hauss$ to $\Delta(f)$ and the complete by continuity.

- Group actions: Every group action can be seen as a functor from a Group to a Set.

  Let G be a group seen as a category with only one element and S be a set seen as a discrete category. Then, an action is a representation of the group of the endomorphism of the set, that is, an action α associate each element of the groupwith an function $\alpha(g) : X \to X$ such that the product of element of the group is maintain via composition. That is for $g \in Ar(G)$ we have that $\alpha(g) \in Ar(X)$ and for

  $$\alpha(g \circ^1 f) = \alpha(g) \circ \alpha(f) \qquad \forall g, f \in Ar(G),$$

where in 1. it is to note that is a group seen as a category the product is the composition) thus having the functoriality.

- `Maybe` Method in Haskell [20, Section 7.1]): In Haskell the definition of Maybe is a mapping from type a to type Maybe a.

```
data Maybe a = Nothing | Just a
```

LISTING 1.1: Declaration of Maybe

Note that `Maybe a` is not a type but a function of types. In order for it to be an endofunctor of the *Hask* category we will need for it to also map function. Let $f : a \to b \in Hask$, then we define the function

$$\text{Maybe}\, f : \text{Maybe}\, a \to \text{Maybe}\, b$$

such that

$$(\text{Maybe}\, f)(Nothing) = Nothing, \qquad (\text{Maybe}\, f)(Just\ a) = Just\ f(a).$$

Note that $(\text{Maybe}\, id_a)(\text{Maybe}\, a) = Nothing \mid Just\ id_a(a) = \text{Maybe}\, a$.

We can now construct the category of all small categories Cat. This category has as object all small categories and as arrows all functor between them. Note that Cat does not contain itself.

We can consider some properties in functors. Note that we can consider a functor $T : C \to D$ as a function over homeset, that is, a function:

$$T : \hom_C(a, b) \to \{Tf : Ta \to Tb | f \in C\} \subset \hom_D(Ta, Tb)$$

Definition 1.2.8. A functor $T : C \to D$ is *full* if it is surjective as a function over homesets, i.e. if the function $T : \hom_C(a, b) \to \hom_D(Ta, Tb)$ is surjective for every $a, b \in C$. A functor is *faithfull* if it is injective over homesets.

As we have defined the concept of opposite category, we can consider functors $T : C^{op} \to D$. This functor, is called a *contravariant* functor from C to D, in oposition of a covariant functor from C to D. A functor $T : C \to D^{op}$ is usually called a *covariant* functor from C to D.

We shall continue defining natural transformation. In the words of Saunders Mac Lane:

> Category has been defined in order to define functor and functor has been defined in order to define natural transformation.

One can see a functor $T : C \to D$ as representation of a category in another, in the sense that a functor provide a picture of the category C in D. Further into this idea, we can consider how to transform these drawings into each other.

Definition 1.2.9. Given two functor $T, S : C \to D$, a *natural transformation* $\tau : T \Rightarrow S$ is a function from $Ob(C)$ to $Ar(D)$ such that for every arrow $f : c \to c' \in C$ the following diagram:

$$\begin{array}{ccc} Tc & \xrightarrow{\tau c} & Sc \\ \downarrow{\scriptstyle Tf} & & \downarrow{\scriptstyle Sf} \\ Tc' & \xrightarrow{\tau c'} & Sc' \end{array}$$

commutes. A natural transformation where every τc is invertible is called a *natural equivalence* and the functors are *naturally isomorphic*.

That is a natural transformation is a map from pictures of C in to pictures of D. Note that a natural transformation acts only on the domain of objects. Lets provide some examples.

When we have to define natural transformations it will often be useful to define each arrow individually. In this case, with the same notation as in the definition, we will note $\tau(c) = \tau_c : Tc \to Sc$.

Example 1.2.3.

- The opposite group: We can define the opposite functor $(\cdot)^{op} : Grp \to Grp$ that maps $(G, *)$ to $(G; *^{op})$ where $a *^{op} b = b * a$ and map a morphism $f : G \to G'$ we define $f^{op}(a) = f(a)$ and have that

  $$f^{op}(a *^{op} b) = f(b * a) = f(b) * f(a) = f^{op}(a) *^{op} f^{op}(b).$$

  Denoting the identity function $Id : Grp \to Grp$, we have a natural transformation $\tau : Id \Rightarrow (\cdot)^{op}$ defined by $\tau_G(a) = a^{-1}$ for all $a \in G$.

- In Haskell, the safeHead [20, Section 10.1] function between the List functor and the Maybe functor. Lets start by defining the functors:

  - We can define the list functor that maps the type a to the type $[a]$ and apply each function $f : a \to b$ to $f' : [a] \to [b]$ that applies f elements wise (on empty list it does nothing).

    ```
    data List a = Nil | Cons a (List a)
    ```

  - We can define the natural transformation SafeHead : List $\to$ Maybe defined as follow:

    ```
    safeHead :: [a] -> Maybe a
    safeHead [] = Nothing
    safeHead (x : xs) = Just x
    ```

1.2.3 Constructions

In this last subsection we will introduce some standard construction on categories, along with some examples of these constructions.

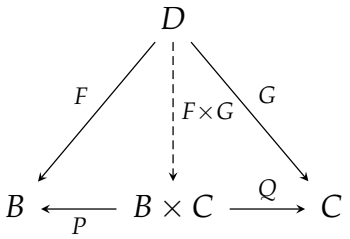
Product Category

We present now one of the most usual construction in mathematics: the product. We will consider the "universal" properties of product on the next chapter. By now, we present the product of categories.

Definition 1.2.10. Let B, C be categories. Then the *product category* $B \times C$ is the category that has as objects the pairs $\{\langle b, c\rangle : b \in Ob(B), c \in Ob(C)\}$ and as arrows the pairs of arrows $\{\langle f, g\rangle : f \in Ar(B), g \in Ar(C)\}$. The composition of arrows is defined by the elementwise composition.

It is clear that we can define to functor $P : B \times C \to B$ and $Q : B \times C \to C$ that restricts the category to each of its component parts (functorial axioms follow immediately). Moreover, we can see that any functor $F : D \to B \times C$ will be uniquely identified by its composition by P and Q.

Complementary, for any two functors $F : D \to B, G : D \to C$ we can define an functor $F \times G : D \to B \times C$ that apply $(F \times G)\langle f, g\rangle = \langle Fg, Gg\rangle$. Expressed as a diagram:



A functor $F : B^{op} \times C \to D$ is called a *bifunctor*. Arguably the most important bifunctor is the $\hom_C$ function seen as a functor. Given a category C we can see $\hom_C : C^{op} \times C \to Set$ as a bifunctor such that:

$$\hom_C(\cdot, \cdot)(a, b) = \hom_C(a, b) \qquad \forall a, b \in Ob(C)$$

for the object. For the arrows:

$$\begin{aligned} \hom_C(f^{op}, g) : \hom_C(a', b) \to \hom_C(a, b') \qquad & \forall f : a \to a', g : b \to b' \in C \\ \hom_C(f^{op}, g)(h) = g \circ h \circ f \qquad & \forall h \in \hom_C(a', b) \end{aligned}$$

From this bifunctor we can define two functors for every $c \in Ob(C)$:

- The functor $\hom_C(c, \cdot) : C \to Set$ such that

$$\hom_C(c, \cdot) = \hom_C(c, d) \qquad \forall d \in Ob(C)$$

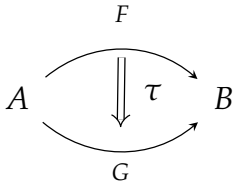
$$\begin{aligned} \hom_C(c, g) : \hom_C(a, d) \to \hom_C(a, d') \qquad & \forall g : d \to d' \in C \\ \hom_C(c, g)(h) = g \circ h \qquad & \forall h \in \hom_C(c, d) \end{aligned}$$

- The covariant functor $\hom_C(\cdot, c) : C^{op} \to Set$, defined analogously.

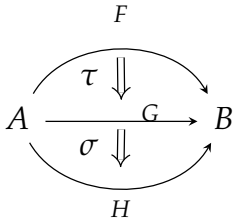
Functor Categories

We continue defining functor categories, that is, categories where we consider the functors as objects and natural transformation as arrows in some sense. This concept will be instrumental in further consideration in the realm of functional programming (in particular, in the definition of monad).

Before explaining composition between natural transformation, it is useful to present the following diagram. Let B, C be categories, $F, G : B \to C$ be functors and $\tau : F \to G$ natural transformation τ, it is common to represent this structure with:



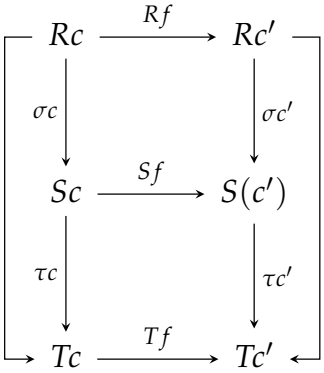
Lets define the composition of two natural transformation. We will start defining the composition of two natural tranformations σ, τ as in:



Due to this representation, this composition of natural transformations is called *vertical composition*, in opposition to the *horizontal composition* (def. 1.2.13).

Definition 1.2.11. Let C and B be two categories, $R, S, T : C \to B$ be functors, and let $\tau : R \to S$, $\sigma : S \to T$, we define the composition $(\tau \circ \sigma)c = \tau c \circ \sigma c$.

To see that $(\tau \circ \sigma)$ is a natural transformation it suffices the following diagram[23]:



Definition 1.2.12. Let B, C be categories. We define the category B^C as the functor category from C to B, that is, the category with all functors $F : C \to B$

as object, natural transformation as arrows, and composition as defined in 1.2.11.

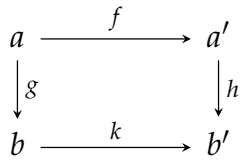
we now present some examples to provide some intuition about when this type of construction are considered.

Example 1.2.4.

- The category of group action over a set: As we have seen in 1.2.2 each group action over a set is a functor. Let G be a group and S be a set, both seen as categories. Then we can consider the category S^G, that has as object every group action of G over S and as arrow the morpishm of actions.

- C^2: In 1.2.1 we define the 2 category and can consider therefore the category of functor from 2 to C. This category is called the arrow category, as the can see that there are a functor from 2 to C as functor for each arrow in C and conversely.

  This example display a interesting idea: we can consider small collection of object as functions/functors. This idea can as well be seen when we define a pair of real numbers as any function $f : \{0, 1\} \to \mathbb{R}$. Should we want, for example, to consider all the squares in a category C, we might ass well define the category square Sqr and consider every functor $F : Sqr \to C$.

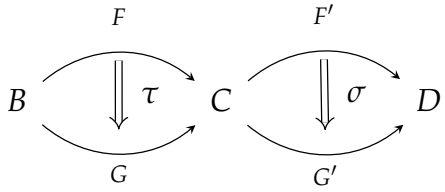


The square category.

  Note that the elements $F : Sqr \to C$ are the arrows of the arrow category of C.

- We can use this construction to study the category C^C of endofunctors of a particular category C. This case is particularly interesting while studying the endofunctors present in $Hask^{Hask}$ and later consider the Monad as a programming structure.

Note that this is not the only way in which to define the composition of natural tranformation. In fact, we can define another functor category. In this case we compose two natural transformation as in:



Lets formalize this composition:

Definition 1.2.13. Let B, C, D be categories, $F, G : B \to C, F', G' : C \to D$ be functors, and let $\tau : F \to G$, $\sigma : F' \to G'$, we define the composition $(\tau \circ \sigma) : F \circ F' \to G \circ G'$ such that

$$(\sigma \circ \tau)c = Fc \circ G'\tau c$$

for all $c \in B$.

With this horizontal composition we can properly defined.In this case we can see that the composition of two natural transformation is indeed a natural transformation due to the commutativity of:

$$\begin{array}{ccc} F'Fc & \xrightarrow{\sigma Fc} & G'Fc \\ \downarrow{\scriptstyle F'\tau c} & \searrow^{(\sigma\circ\tau)c} & \downarrow{\scriptstyle G'\tau c} \\ F'Gc & \xrightarrow{\sigma Gc'} & G'Gc' \end{array}$$

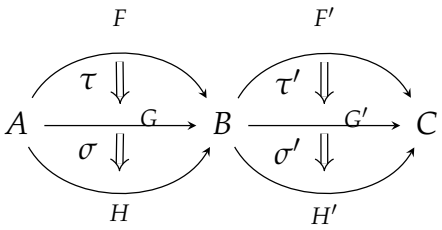
With this definition of composition we can consider another different category: the category of all functors of all (small) categories, that is, the category that has all functors as object, and has the natural transformation with horizontal composition as arrows.

When we have to consider both compositions at the same time we denote the vertical composition with $\tau \cdot \sigma$ and horizontal composition with $\tau \circ \sigma$, as in [19]. Lastly we have to consider how this composition relate to each other. This is seen in the *interchange law*:

Proposition 1.2.3. Let A, B, C be categories, $F, G, H : A \to B$, $F', G', H' : B \to C$ be functors and $\tau : F \to G, \sigma : G \to H, \tau' : F' \to G', \sigma' : G' \to H'$ be natural transformations, then:

$$(\sigma' \circ \sigma) \cdot (\tau' \circ \tau) = (\sigma' \cdot \tau') \circ (\sigma \cdot \tau)$$

Proof. We have a structure like



From the naturality of τ' we have that for all $c \in A$:

$$\begin{aligned}((\sigma' \cdot \tau') \circ (\sigma \cdot \tau))(c) &= H'((\sigma \cdot \tau)c) \circ (\sigma' \cdot \tau')(Fc) \\ &= H'(\sigma c \circ \tau c) \circ (\sigma'(Fc) \circ \tau'(Fc)) \\ &= H'(\sigma c) \circ H'(\tau c) \circ \sigma'(Fc) \circ \tau'(Fc) \\ &= H'(\sigma c) \circ \sigma' G c \circ G' \tau c \circ \tau'(Fc) \\ &= (\sigma' \circ \sigma)(c) \circ (\tau \circ \tau)(c) \\ &= ((\sigma' \circ \sigma) \cdot (\tau' \circ \tau))(c).\end{aligned}$$

□

To finally consider the relation between products, and the functor category we can see that given 3 small categories A, B, C we have a bijection:

$$\hom_{Cat}(A \times B, C) \cong \hom_{Cat}(A, C^B).$$

This fact will be taken into account further in the text, as this will mean that the functor $\cdot \times B : Cat \to Cat$ has a right adjoint.

Comma Category

To define the comma category, we define the Category $(b \downarrow S)$ of object S-under b, sketch the dual notion, and generalize this two concepts with the comma category.

Given a functor $F : B \to C$ an object $b \in B$ is $F-$under another object $c \in C$ if there exists an arrow $f : c \to Fb \in C$. This can be represented as

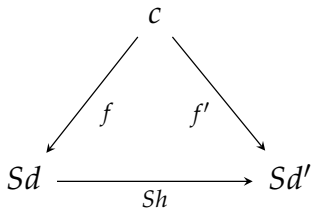
$$\begin{array}{c} c \\ \downarrow f \\ Fb \end{array}$$

and thus the name of F-under.

Definition 1.2.14. Let B, C be (small)[1] categories and $S : B \to C$ be a functor. For every $c \in Ob(C)$ we can define the category $(c \downarrow S)$ that has as objects all pairs $(b, f) \in Ob(B) \times Ar(C)$ such that $f : c \to Sb$ and as arrows $h : (d, f) \to (d', f')$ every arrow $h : d \to d' \in Ar(B)$ such that $f' = Sh \circ f$.

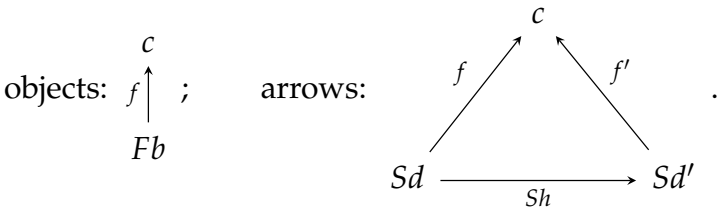
[1] check this

The property that each arrow should satisfy can be represented as:



Example 1.2.5. Let $U : Grp \to Set$ be the forgetful functor, and let $X \in Ob(Set)$. We can consider $(X \downarrow U)$ where every object is a function $f : X \to Ug$ for a group g.

One can easily now deduce the dual concept of the category $(b \uparrow S)$ represented by:



Now suppose that we have three categories B, C, D and to functors S, T such that

$$B \xrightarrow{S} C \xleftarrow{T} D$$

me might want to consider the relations objects of B and D, for that we have the comma category:

Definition 1.2.15. Let B, C, D be categories and to $S : B \to C, T : C \to D$. We define the coma category $(S \downarrow T)$ as the category that has as object the triples (b, d, f) with $b \in Ob(B), d \in Ob(D), f : Sb \to Td \in Ar(C)$ and arrows the pairs $(g, h) : (b, d, f) \to (b', d', f')$ with $g : b \to b' \in Ar(B), h : d \to d' \in Ar(D)$ such that $Th \circ f = f' \circ Sg$.

We can represent the previous definition by:

The name comma category comes from is alternative notation $(S \downarrow T) = (S, T)$. We prefer the $(S \downarrow T)$ as it is more clear, nonetheless it is clear that before modern text editor became popular, the original comma notation had a big plus.

Chapter 2

Universality, Adjoints and Closed Cartesian Categories

TODO Small paragraph introducing the notions, and sketching a brief summary of the chapter.

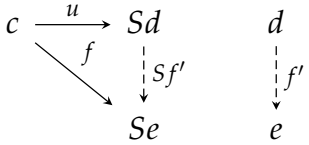
2.1 Universality

In this section we present the concept of universality. This concept is behind lots of mathematical properties. Intuitively, universality is an efficient way of expressing an one-to-one correspondence between arrows of different categories. This one-to-one relationship is usually expressed via "given an arrow y it exists one and only one arrow x such that <insert your favorite universal property>".

Prior to the formal definition, we shall introduce an example. Probably the first contact that any mathematician has with universality is when we first try to define a function $f : \mathbb{R} \to \mathbb{R}^2$. We quickly understand that defining such a function is equivalent to define two $g, h : \mathbb{R} \to \mathbb{R}$(we further explain the product in 2.1.6). This uniqueness is the flavor that attempts to capture the concept of universality. Other examples of unique existence are those that occur in quotient groups or in bases of vector spaces. This example will be further formalized after the definition.

Definition 2.1.1. Let $S : D \to C$ be a functor and $c \in Ob(C)$, an *universal arrow* from c to S is a pair (d, u) with $d \in Ob(D), u : c \to Sd \in Ar(C)$, such that for every (e, f) with $e \in Ob(D)$ and $f : c \to Sd$ there exists an unique $f' : r, d \in Ar(D)$ such that $u \circ Sf' = f$.

In a diagram:



Note that an universal arrow (d, u) induces the unique existence of an arrow in D, but with interesting properties via it relationship with S. Whenever possible to provide an universal arrow we will only define the functor $S : D \to C$ and the arrow $u : c \to Sd$, letting all other information be deduced from the context.

Lets formalize the prior examples and provide some more:

Example 2.1.1.

- Quotient Group:

  From this property alone the three isomorphism theorems can be deduced. Therefore, we only have to prove this result to have the full power of these theorems in any context (e.g. Rings, K-Algebra, or Topological spaces).

- Product in $\mathbb{R}$:

- Vector Space Bases:

- Haskell Type void (or some other cool example):

Lastly, we will provide a characterization of universality:

Proposition 2.1.1. Let $S : D \to C$ be a functor and $u : c \to Sr \in Ar(C)$. Then u is an universal arrow if and, only if, the function $\varphi : \hom_D(r, \cdot) \to \hom(c, S\cdot)$ such that $\varphi(d)(f) = Sf \circ u$ for all $f \in \hom_D(r, d)$ is a natural bijection. Conversely, every natural bijection is uniquely determined by an universal arrow $u : c \to Sr$.

Proof. Lets start by supposing that u is universal. Then for φ to be universal the diagram

$$\begin{array}{ccc} \hom_D(r,d) & \xrightarrow{\varphi(d)} & \hom_C(r, Sd) \\ {\scriptstyle \hom_D(r,g)}\downarrow & & \downarrow{\scriptstyle \hom_C(r,Sg)} \\ \hom_D(r,d') & \xrightarrow{\varphi(d')} & \hom_C(r, Sd') \end{array}$$

should commute for all $g \in Ar(D)$. As this is a diagram in the category of sets, we can check the commutativity by checking it element wise. For any $f \in \hom_D(r, d)$:

$$\begin{array}{ccc} f & \xrightarrow{\varphi(d)} & Sf \circ u \\ {\scriptstyle \hom_D(r,g)}\downarrow & & \downarrow{\scriptstyle \hom_C(r,Sg)} \\ g \circ f & \xrightarrow{\varphi(d')} & S(g \circ f) \circ u = Sg \circ Sf \circ u \end{array}$$

So the diagram commutes, and φ is natural. The bijectivity follows from the definition of u being universal.

Lets consider now that φ is a natural bijection. We will define $u := \varphi(r)(1_r)$ and check that (r, u) is an universal arrow. As φ is natural we have that:

$$\begin{array}{ccc} \hom_D(r,r) & \xrightarrow{\varphi(r)} & \hom_C(r, Sr) \\ {\scriptstyle \hom_D(r,g)}\downarrow & & \downarrow{\scriptstyle \hom_C(r,Sg)} \\ \hom_D(r,d) & \xrightarrow{\varphi(d)} & \hom_C(r, Sd) \end{array}$$

Writing the diagram for the element $1_r \in \hom_D(r,r)$ and any $d \in Ob(D), g : r \to d \in \hom_D(r,d)$:

$$\begin{array}{ccc} 1_r & \xrightarrow{\varphi(r)} & u \\ {\scriptstyle \hom_D(r,g)}\downarrow & & \downarrow{\scriptstyle \hom_C(r,Sg)} \\ g & \xrightarrow{\varphi(d)} & \varphi(d)(g) = Sg \circ u \end{array}$$

and since φ is a bijection, for every $f \in \hom_C(r, Sd)$ there is an unique function $f' = \varphi(d)^{-1}(f)$ such that $Sf' \circ u = f$, thus being u universal.

□

From this theorem there is a definition that arises:

Definition 2.1.2. Let D be a category with small home-sets and let $F : D \to C$ be a functor. A representation of a functor $K : D \to Set$ is a pair (r, φ) with $r \in Ob(D)$ and φ a natural isomorphism such that

$$D(r, \cdot) \equiv_\varphi F.$$

A functor is representable if it has a representation.

Note that therefore a universal arrow induces a natural isomorphism $D(r,d) \equiv C(c, Sd)$ and this induces a representation of the functor $C(c, S\cdot) : D \to Set$.

2.1.1 Yoneda's lemma

In this subsection deals with the Yoneda's Lemma. Mac Lane[19] assures the lemma first appeared in his private communication with Yoneda in 1954. With time, this result has became one of the most relevant one in Category Spaces. We will start by providing some intuition to it, followed by it proof and some use cases.

This results is due to japanese professor Nobuo Yoneda. We know about Yoneda's life thanks to the elegy that was written by Yoshiki Kinoshita[25]. Yoneda was born in Japan in 1930, and received his doctorate in mathematics from Tokyo University in 1952. He was a reviewer for international mathematical journals. In addition to his contributions to the field of mathematics, he also devoted his research to computer science.

The idea behind the yoneda lemma can be arid at first, if one does not have a prior understanding of what the purpose and usefulness of this lemma is. In order to ilustrate this idea we will introduce a (simplified) definition of Moduli Spaces, so that we have a geometric understanding of Yoneda's Lemma.

The idea behind (some) Moduli spaces is to classify algebraic curves up to isomorphisms. In addition, Moduli spaces allow us to control "complicated" mathematical objects (such as a quotient space of unknown objects) by simpler objects or objects with better properties (such as a concrete variety). A canonical example of this type of classification is:

$$\{\text{Vector spaces of finite dimension}\}/\text{isomorphism} \cong \mathbb{N}$$
$$[V] \to dimV.$$

Where a complex object can be classified by another object of which we know more properties. We can start by defining:

$$\mathcal{M} = \{\text{smooth complex non singular curves}\}/\text{isomorphism}$$

Further on when we talk about curves we will refer to smooth complex non singular curves. Note that if two curves are isomorphic then they have the same genus. Therefore the function

$$\gamma : \mathcal{M} \mapsto \mathbb{N}$$
$$[V] \mapsto \text{genus of V}$$

is well defined and we can define $\mathcal{M}_g = \gamma^{-1}(g)$. An interesting classification of $\mathcal{M}_g$ is given when we consider that for every g there exists a closed, connected, non-singular variety U_g and a family $\{C_t : t \in U_g$ such that a curve of genus g will be a fibration of C_t. Moreover there is a variety M_g and a superjjective morphism $\varphi : U_g \to M_g$ such that $\varphi(t_1) = \varphi(t_2)$ if $C_{t_1} \equiv C_{t_2}$. Therefore we are classifying the equivalence classes of $\mathcal{M}_g$ by points of the variety M_g (thus generating a Moduli problem).

Similarly to this two example, with the Yoneda lemma we will have a functor $F : D \to Set$ and one representation of this functor. We will classify the natural transformation of these functors by the set in the image of F! Interestingly enough, there will be applications where the complex object is not the space of naturals transformations, but the images of F (see 2.1.2 for an example). Lets proceed to enunciate and proof the result.

Theorem 2.1.1. *[19, Section 3.2] Let D be a category with small home-sets, $F : D \to Set$ be a functor, and $r \in Ob(D)$. Then there is a bijection*

$$\tau : Nat(\hom_D(r, \cdot), F\cdot) \equiv Fr$$
$$\tau(\alpha : \hom_D(r, \cdot) \to F\cdot) = \alpha(r)(1_r)$$

Where τ is natural in K(as an object of Set^D) and in r.

Proof. As α is a natural transformation we have that, for every $g : r \to d \in Ar(D)$:

$$\begin{array}{ccc} \hom_D(r,r) & \xrightarrow{\alpha(r)} & Kr \\ {\scriptstyle \hom_D(r,g)}\downarrow & & \downarrow{\scriptstyle Kg} \\ \hom_D(r,d) & \xrightarrow{\alpha(d)} & Kd \end{array}$$

Writing $\alpha(r)(1_r) = u$ we have that:

$$\begin{array}{ccc} 1_r & \xrightarrow{\alpha(r)} & u \\ {\scriptstyle \hom_D(r,g)}\downarrow & & \downarrow{\scriptstyle Kg} \\ g & \xrightarrow{\alpha(d)} & \alpha(d)(g) = Kg(u) \end{array}$$

Therefore every natural transformation is uniquely identified by the value of u, therefore τ is injective. Moreover, for every u in Kr, we can define a natural transformation following the previous diagram, therefore, τ is bijective.

To see that τ we is natural we have to consider for which functor it is natural. Consider the functor *Evaluation* $E : Set^D \times D$ that maps each $(F, c) \to Fc$, and the functor $N : Set^D \times D \to Nat(\hom_D(r, \cdot), K)$ the set of natural transformations. Finally, $\tau : N \to E$ is a natural transformation. □

The first functor that we will want to apply this result is to the hom. functor. But this functor is a bifunctor, so to get the full result of this lemma applied to the full bifunctor we may restate this lemma to contravariant functors.

Corollary 2.1.1.1. *Let D be a category with small home-sets, $F : D \to Set$ be a contravariant functor, and $r \in Ob(D)$. Then there is a bijection*

$$\tau : Nat(\hom_D(\cdot, r), F\cdot) \equiv Fr$$
$$\tau(\alpha : \hom_D(\cdot, r) \to F\cdot) = \alpha(r)(1_r)$$

Where τ is natural in K(as an object of Set^D) and in r.

Proof. We seek to use the Yoneda lemma in the functor $F' : D^{op} \to Set$ induced by F. Then we have that:

$$\tau : Nat(\hom_{D^{op}}(r, \cdot), F'\cdot) \equiv F'r$$
$$\tau(\alpha : \hom_{D^{op}}(r, \cdot) \to F'\cdot) = \alpha(r)(1_r)$$

Taking into account that $F_{|Ob(D)} = F'_{|Ob(D^{op})}$, and that $\hom_{D^{op}}(r, \cdot) = \hom_D(\cdot, r)$ we have the result. □

As we have seen, the Yoneda lemma is a direct generalization of the moduli problem. In the same vein, yoneda's lemma is the generalisation of other problems/theorems in mathematics, most notably Cayley's lemma. It states:

Proposition 2.1.2. Any group is isomorphic to a subgroup of a symmetric group.

To understand this, take a groups G seen as a single-object category, and name that object e. Then, the functor $\hom_G(e,\cdot) : G \to Set$ can be seen as a group action 1.2.2. Then the Yoneda lemma states that:

$$Nat(\hom_G(e,\cdot), Hom_G(e,\cdot)) \equiv_\varphi Hom_G(e,e).$$

Translating this result to group theory:

- Remember that $Hom_G(e,e)$ is the group G.
- Every natural transformation is a equivariant map between G-sets.
- This equivariant maps, forms an endormophism group under composition, being a subgroup of the group of permutations.
- This natural isomorphism φ define a group isomorphism.

So we have the isomorphism of groups that is stated in Cayley's Theorem.

We continue our exploration of Yoneda lemma by defining the *Yoneda Embedding*. For that we define the contravariant functor $h_a = \hom_C(\cdot, a)$. Then the contravariant Yoneda lemma tell us that:

$$Nat(h_a, h_b) \equiv_{\tau_a} \hom(a,b).$$

We then can define a fully faithfull embedding $v : C \to Set^{C^{op}}$ such that

$$\begin{aligned} va &= \hom_C(A,\cdot) && \forall a \in Ob(C), \\ vf &= \tau_a^{-1}(f) && \forall f : b \to a \in Ar(C). \end{aligned}$$

This functor allows us to view the category C as a subcategory of the category of contravariant functors from C to Set, which will be useful for determining "heritable" properties in C.

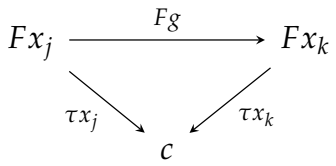
2.1.2 Properties expressed in terms of universality

After the examples given, we will define a few constructions that are present in various parts. We will outline the notions of limit, pullback and product, and the dual notions of colimit, pushout and coproduct.

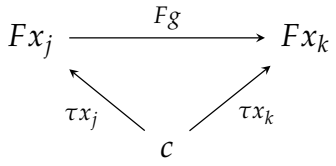
The notions of product and pullback can be seen as particular cases of the notion of limit. We will therefore begin by defining this concept as an introductory step. In turn, to define limit we will introduce the concept of co-cone and the diagonal functor.

Definition 2.1.3. Let C, J be categories. We can define the functor $\Delta_J : C \to C^J$ that maps c to the functor from J to C that is constantly c, and maps every arrow to the identity 1_c.

Whenever possible we will write only Δ, and let the information of the category be deduced from context. J is usually small and often finite. We can now consider a natural transformation $\tau : F \to \Delta c$. This can be represented as in the following diagram:



commutes for every $g : x_j \to x_k \in Ar(j)$, for that reason, such natural transformation is usually called a co-cone. The dual notion is called cone and is represented as:

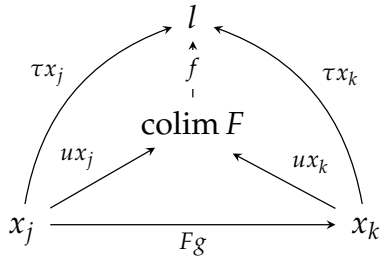


We can now define the concept of limit and colimit. We introduce first the concept of colimit. This definition is a basic definition of a universal arrow, only in a category of functors. Following this definition, we will define the limit as a dual concept.

Definition 2.1.4. A colimit is an object $r \in Ob(C)$ together with an universal arrow $u : F \to \Delta r \in Ar(C^J)$. The colimit is denoted by

$$\varinjlim F = r = \operatorname{colim} F.$$

The notation $\lim_{\leftarrow}$ is intuitive as we can see that in the colimit we have arrows to F. To represent this as a diagram, we have a co-cone $u \to \operatorname{colim} F$ such that for every other co-cone $\tau \to s$, it exist an unique f such that the following commutes for every $x_j, x_k \in Ob(C)$:

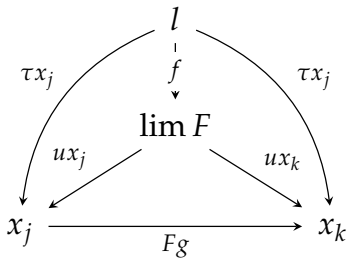


Now, thanks to the duality of categories, we can define what a limit is in a very synthetic way:

Definition 2.1.5. A limit is the dual concept of a colimit. It is denote as

$$\varprojlim F = r = \lim F.$$

A limit is represented by the following diagram:



Analogously as in the colimit notation, in the limit we have arrows from F, and thus the notation $\lim_{\rightarrow}$. Let focus for a while now in the notion of limit, in particular of two of its specials cases: the product and the pullback. From these cases we are going to provide most examples.

We have already talked about the product of categories, and in example 2.1.1 we denote that in these type of construction there is some sort of universality involved. The product is a limit when J is the 2-element discrete category, that is, when every functor from $J \to C$ is merely choosing to object of C.

Definition 2.1.6. Let C be a category, $J = \{0, 1\}$ be the discrete category with two elements. The product $c_1 \times c_2$ of two elements $c_0, c_1 \in Ob(C)$ is the limit of the functor $F : J \to C$ such that $F0 = c_0, F1 = c_1$.

This construction means that providing an arrow to c_0, c_1 determines an unique arrow to $c_0 \times c_1$. In this case, the arrows $u0, u1$ are usually called *projections* and denoted by π_0, π_1. The notation is due to the product being a generalization of the cartesian product in the category *Set*. It is also sometimes note with $c_0 \Pi c_1$ with Π being stardard for the sequence product. Some examples of product object in categories are:

Example 2.1.2.

- Product of Banach spaces:
- Product of something in haskell.

Analogously, we can define the coproduct, on which instead of defining an arrow to c_0, c_1, we define an object *from* c_0, c_1.

Definition 2.1.7. The coproduct is the dual definition of the product. It is denoted by $c_0 \sqcup c_1$.

In this notation $\sqcup$ denotes an inverted Π, with the meaning of being the dual notion of the product.

Example 2.1.3.

- Tensor Product of R-Modules, include cite to Rotman-Theorem 1.5.

- Union of enumerate types.

From my personal experience I have to say that I have seen more difficulties learning this notion rather than learning the utterly similar notion of product, probably because we are more used to think in terms of arrays rather than in terms of universality for the product. I think that thinking in terms of universality should be the way in to these concept.

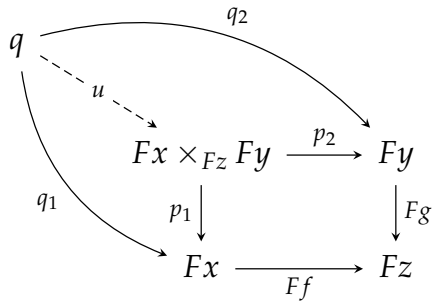
After learning about the product and the coproduct, we will focus now in the notion of pullback and its dual, the pushout. We will first define the category

$$P = \quad x \xrightarrow{f} z \xleftarrow{g} y$$

Then we can define the pullback:

Definition 2.1.8. Let C be a category and $F : P \to C$ be a functor. Then the pullback of Fx and F_y denoted as $Fx \times_{Fz} Fy$ is the limit of the functor F.

We can represent this structure in the following diagram. For any other object q and arrows $f' : q \to Fx, g' : q \to Fy$ we have:



Analogously, we can define:

$$CoP = \quad x \xleftarrow[f]{} z \xrightarrow[g]{} y$$

and define:

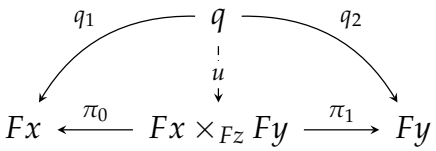
Definition 2.1.9. Let C be a category and $F : CoP \to C$ be a functor. Then the pushout of Fx and F_y denoted as $Fx \sqcup_{Fz} Fy$ is the colimit of the functor F.

Example 2.1.4.

- Fiber bundles (pullback)
- Suppose that X, Y, and Z as above are sets, and that f : Z $\to$ X and g : Z $\to$ Y are set functions. The pushout of f and g is the disjoint union of X and Y, where elements sharing a common preimage (in Z)
- Seifert-Van-Kampen (Si tienes valor).

We can regard that there are an evident similarity on the notation of the pullback/pushout and the one of product/coproduct. To understand these we have to consider the similarities of both construction. We are going to focus on the similarities of product and pullback.

In both case, the universal property consists of having an arrow to a generated object only if we have an arrow to each of its generators. In this line of reasoning, we can consider that the product is a pullback where we forget about the object z and its arrows. One easy way to generate that case is to consider the construction when Fz is a terminal object. In that case we have that the existence of Ff and Fg is a tautology, and we can only consider



having a product structure. One can proceed to consider an analogous consideration with pushout and coproduct.

2.2 Adjoints

The adjoints is a fundamental property on Category theory. It relates two functors $F : C \to D, G : D \to C$. The importance of adjoints however, comes from it ubiquity among mathematics, often relate with the ubiquity of universality. This notion was first presented by Daniel M. Kan in [14]. For this section we follow [19].

We will start by providing some intuitive notions, before a formal definition. Take the Forgetful Functor $U : Grp \to Set$, and the Free Group Functor $\mathcal{F} : Set \to Grp$. We can see that it is not difficult to consider that we can compose F and G to make endofunctors. These endofunctors are far from being identities, nonetheless we have still a great relation between them: for every set X and group G, there is a function $X \to UG$ for each morphism $FX \to U$. Conversely, any function from X to UG induces a morphism.

Any avid reader will detect by this point the taste of universality. But there is a bit more, we have a bijective relation on every image home-set of both F and G. This is the underlying property of adjointness - to relate home-sets. Lets formalize this idea:

Definition 2.2.1. Let C, D be categories. And adjunction is a triple $(F, G, \varphi) : D \to C$ where F, G are functors:

$$D \underset{F}{\overset{G}{\rightleftarrows}} C$$

while φ is a transformation that maps each $(c, d) \in Ob(C \times D)$ with a bijection:

$$\varphi_{c,d} : \hom_C(Fd, c) \equiv \hom_D(d, Gc).$$

which is natural in c and d.

We have a bit to unpack in this definition. We will start to understand in what sense φ is natural. Remembering about N functor in Yoneda's Lemma, we can see that it is natural bijection from $\varphi : Hom_C(F\cdot, \cdot) \to \hom_D(\cdot, G\cdot)$ in each variable. That is, for every $f : d \to d' \in C, k : c \to c' \in D$ the diagrams

$$\begin{array}{ccc} \hom_C(Fd,c) & \xrightarrow{\varphi_{c,d}} & \hom_D(d,Gc) \\ \downarrow{\scriptstyle \hom_C(Fd,\cdot)(g)} & & \downarrow{\scriptstyle \hom_D(d,G\cdot)(g)} \\ \hom_C(Fd,c') & \xrightarrow{\varphi_{c',d}} & \hom_D(d,Gc') \end{array} \qquad \begin{array}{ccc} \hom_C(Fd,c) & \xrightarrow{\varphi_{c,d}} & \hom_D(d,Gc) \\ \downarrow{\scriptstyle \hom_C(F\cdot,c)(f)} & & \downarrow{\scriptstyle \hom_D(\cdot,Gc)(f)} \\ \hom_C(Fd,c') & \xrightarrow{\varphi_{c,d'}} & \hom_D(d,Gc') \end{array}$$

comutes.

We can follow by considering the important notation decision: how to call the adjoints. This is a quite ambiguous notation, but we will stick with the notation of F being the *left adjoint* and G being the *right adjoint*. When a functor *is* a left (resp. right) adjoint it *has* a right (resp. left) adjoint. This notation comes from where is the functor placed in the home-set, when making the bijection.

We have already suggested the relationship between universality and adjointness. We can summary that relation in the following property:

Proposition 2.2.1. An adjunction $(F, G, \varphi) : D \to C$ determines:

- a natural transformation $\eta : I_D \to GF$ such that $\eta(x) : x \to G$ is universal for every $x \in Ob(D)$. Conversely, we can define:

$$\varphi(f : Fd \to c) = Gf \circ \eta(d) : d \to Gc.$$

- a natural transformation $\epsilon : FG \to I_C$ such that $\varepsilon(x) : Fx \to a$ is universal for every $x \in Ob(C)$. Conversely, we can define:

$$\varphi(g : d \to Gc) = \eta(c) \circ Fg : Fd \to c.$$

Proof. • Let $a = Fd$, then we have that

$$\hom_C(a,c) \equiv \hom_D(d,Gc).$$

and by proposition 2.1.1 we have an universal arrow $\eta(d) : d \to Ga = GFd$ is provided. The function $d \to \eta(d)$ is natural $I_D \to GF$ as:

$$\begin{array}{ccc} d & \xrightarrow{\eta c} & GFd \\ \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle GFh} \\ d' & \xrightarrow{\eta c'} & GFd'' \end{array}$$

commutes.

- Analogous.

□

From this proposition we can see that we may be able to define an adjoint based only on the universal arrows provided. We can summary a few equivalent definition of adjoint:

Proposition 2.2.2. Each adjoint $(F, G, \varphi, \eta, \varepsilon) : D \to C$ is completely determined by the items in any one of the following list:

1. F, G and a natural transformation $\eta : 1_D \to GF$ such that $\eta(d)$ is universal for every $d \in (D)$.
2. F, G and a natural transformation $\varepsilon : FG \to 1_C$ such that $\varepsilon(c)$ is universal for every $c \in (C)$.
3. G and for each $d \in Ob(D)$:
   - an object $F_0(d) \in Ob(C)$.
   - an universal arrow $\nu(x) : d \to GF_0 d$.

   Then F has F_0 as object function and $F(h : d \to d') = \eta(x') \circ h$.
4. F and for each $c \in Ob(C)$:
   - an object $G_0(c) \in Ob(D)$.
   - an universal arrow $\varepsilon(c) : c \to GF_0 c$.
5. F, G and two natural tranformation $\eta : I_D \to GF, \varepsilon : FG \to I_C$ such that both composites are identities.

Proof. 1. We need to define the natural bijection $\varphi : C(Fd, c) \to D(d, Gc)$. Defining $\varphi(g) = Gg \circ \eta(c)$ for each $g : Fd \to c$. Is is well define as $Gg \circ \eta(c) : d \to Gc$ and it is a bijection due to $\eta(x)$ universality.

It is natural in d because η is natural and is natural in c beacuse G s a functor.

2. Dual of the previous.

3. We will proof that there is only one functor with F_0 as object function such that $\eta(x) : I_D \to GF$ is natural. The naturality and universality told us that each $h : d \to d' \in Ob(D)$ induces two arrows:

$$\begin{array}{ccc} F_0 d & & d \xrightarrow{\eta(d)} GF_0 d \\ \vdots & & \downarrow h \quad\quad \vdots \\ F_0 d' & & d' \xrightarrow{\eta(d')} GF_0 d' \end{array}$$

So the only way to define $Fh = \eta(x) \circ h$. We conclude applying 1.

4. Dual of the previous.

□

To further ilustrate this concept we present some examples:

Example 2.2.1.
- Free group and forgetful functor
- Čech compactification.

TODO? Lastly, we are going to study the and prove a theorem that allow us to include parameters in adjoints.

2.2.1 Equivalente of Categories

We can define an isomorphism of categories the same way that we define isomorphisms in any other category: an isomorphism is an arrow (Functor) with a two sided inverse. This definition, while standard, is quite restrictive. We are going to need some more lax concept of equivalence of categories, in the sense that while not being exactly the same, they are mostly the same. Formally:

Definition 2.2.2. A functor $F : C \to D$ is an equivalence of categories if there exists a functor $G : D \to C$ and two natural isomorphisms $\varphi : I_C \to GF$ and $\zeta : I_D \to FG$

For the canonical example we may introduce a really organic concept: the *skeleton* of a category. Quite often in mathematics we do not consider all objects of certain type, but rather objects up to isomorphism. The skeleton category of a category C is another category where we consider the objects up to isomorphism. Formally:

Definition 2.2.3. Let C be a category. The skeleton of C, namely $ske(C)$, is a subcategory such that every object in C is isomorphic to an object in $ske(C)$.

The definition of equivalence of categories previously outline will allow us, for example, to consider $C \sim ske(C)$, while not being isomorphic. Continuing our discussion, it is not difficult to draw similarities between equivalences and adjoints. That motivates the further definition:

Definition 2.2.4. Let $(F, G, \varphi, \eta, \varepsilon)$ be an adjoint. It is called an *adjoint equivalence* whenever η and ε are natural equivalences.

It is clear that every adjoint equivalence induces two equivalences in F and G. We state this idea in the following proposition:

Proposition 2.2.3. [19, Theorem 1, 4.4] Let $F : C \to D$ be a functor. Then the following properties are equivalent:

i) F is an equivalence of categories.

ii) F is part of an adjoint equivalence.

iii) F is full and faithfull, and each object $c \in C$ is isomorphic to Sa for some object $a \in A$.

Proof.

ii) $\implies$ *i)* Given an equivalent adjoint $(F, G, \varphi, \eta, \varepsilon)$, then F is an equivalence with G, η, ε.

i) $\implies$ *iii)* $\varphi : GF \equiv I_C$ implies that for every $c \in Ob(C)$ we can consider $c \equiv GFc$. Then considering the natural isomorphism $\zeta : FG \equiv I_D$ states for every $f : a \to a' \in D$

$$\begin{array}{ccc} FGa & \xleftarrow{\zeta^{-1}a} & a \\ \downarrow{\scriptstyle FGf} & & \downarrow{\scriptstyle f} \\ FGa' & \xrightarrow{\zeta a'} & a' \end{array}$$

there is one FGf for each f and we get that G is faithful. Analagously we can see that F is also faithful. To see that G is full we can

iii) $\implies$ *ii)* We can see that

□

In the next few subsection, we are going to introduce categories, with some additional structures onto them. This type of considerations, are commonplace in category theory, and will provide useful considerations.

2.2.2 Monad

2.2.3 Closed Cartesian Categories

The notion of product seen in 2.1.6 is a notion that seek to catch the essence of what the Cartesian product is in the category of Set. In this category it happens that for any two objects, one can consider the product without any hesitance about it existence. A *cartesian category* will be a category that, in some sense, maintain this property (of being closed under catersian product). Formally:

Definition 2.2.5. A cartesian cartesian is a category for which every finite product exists.

Nonetheless, further on the text we will need categories with some even nicer properties. These categories are called *closed catersian categories*. We will define them formally first, and after unpack the information that the definition contains:

Definition 2.2.6. A C is closed cartesian if each of the following functors:

$$\begin{array}{l} F_1 : C \to 1 \\ \quad c \to e \end{array} \qquad \begin{array}{l} F_2 = \Delta : C \to C \times C \\ \quad c \to (c, c) \end{array} \qquad \begin{array}{l} F_3^b : C \to C \\ \quad c \to c \times b \end{array}$$

has a *specified* right adjoint, denoted by:

$$\begin{array}{lll} G_1 : 1 \to C & G_2 : C \times C \to C & G_3^b : C \to C \\ \quad e \to t & \quad (c, b) \to c \times b & \quad c \to c^b \end{array}$$

These adjoints provide us with a lot of information:

- From F_1 being an adjoint we get that t is terminal as for every $c \in Ob(C)$:

$$\{id_e\} \equiv \hom_1(F_1(c), e) \equiv \hom_C(c, Ge = t)$$

- From F_2 we get that every pair c, b has a product at is states the universal property of product:

$$\hom_{C\times C}(F_2(c), (c', b')) \equiv \hom_{C\times C}((c, c), (c', b')) \equiv \hom_C(c, c' \times b')$$

  that is, for every object in $c \in C$, to provide a morphism to $c \to c' \times b'$ is the same that providing to morphisms $c \to c', c \to b'$.

- First of all, as every product exists, and it specified by F_2, we can define F_3^b for every b. Then, it adjointness states that:

$$\hom_C(c \times b, d) \equiv \hom_C(c, d^b)$$

  This object d^b is called the exponential object or map object. The intuitive notion is that this object represent in some way the arrows from b to c, being a generalization of the function set of two small sets. The best way to understand this adjoint is by the natural transformation $\varepsilon^b(c^b \times b) = c$. In this context, this is called *evaluation arrow*. In the context of *Set* this amount to the classic evaluation of a function.

  actually it is a technical condition that this adjoint is natural in b (cita maclane).

Lets provide some examples:

Example 2.2.2.

- *Set*
- The category of compactly generated Hausdorff spaces.
- We will prove later that lambda calculus is, in some sense, a closed cartesian category.

Part II

Lambda Calculus

Chapter 3

Lambda Calculus

In this we will define the lambda calculus as a formal language of computation, and on this define the concept of typing, ubiquitous in programming languages. We will carry out this project raising as many bridges between category theory and the lambda calculus as possible.

The main references for this chapter are [22] for the general approach and concepts relatives to untyped calculus, [18] for more specifics details and the category theory approach to the calculus. We refer to [3] to further understand the historical motivation for the development of the theory.

One of the important advances in 19th century mathematics is the development of the concept of function. However, what makes two functions the same? The most widespread view is the *extensional* approach. In this, we consider two functions to be the same if for the same input they have the same output. In this case, the functions exist by themselves, and each f as a pairing of an X domain to an Y codomain. These functions can be considered as sets $f \subset X \times Y$.

Despite that, specially those involved in computing the functions, can consider this notion of equality misleading. For example, consider a prime p big enough, and let f and g be two endomorphisms of $\mathbb{Z}/p\mathbb{Z}$. One could argue that the endomorphism of $f(x) \to x^2$ is different to $g(x) \to \log_a a^{x+2}$. Despite having the same output for the same input, one is clearly different in complexity. It even involve the resolution of a discrete logarithm, that is highly non trivial.

This view, in which not only is the result of the function important but how is that result obtained, is called the *intensional* approach. This approach, gain traction in early 1930 as Church[5], Gödel[1] or Turing[24] start formalizing what is to be computable. This approach is now as ubiquitous modern computer program care not only on the correctness of a program, but on its time and space complexity (to the point on which correctness is often partially dropped to easy time complexity, for example [12]).

In lambda calculus we consider function as formulas, and thus, we consider an intensional approach. This intensional approach was also backed by the constructivist, to whom we will later on relate with the study of the

Curry-Howard isomorphism [13].

3.1 Untyped Lambda Calculus

We will start defining the most simple version of lambda calculus: untyped lambda calculus. Let's define some concepts:

1. An alphabet A is an arbitrary, maybe finite, non-empty set.
2. A symbol a in an element of the alphabet.
3. A word/formula is a finite sequence of symbols.
4. The collection of all possible words over an alphabet A is denoted by A^*.
5. A language L over A is a subset of A^*.

There are a lot of languages. For example, Spanish is a language, with a well-known alphabet L, with a proper subset of words over L^*.In the same fashion, we define lambda calculus as a formal language, defining its syntax, that is, what words are valid. After that, we will provide the semantics of the language, that is, we will provide meaning to the words.

Syntax of untyped λ-calculus

We start with the basic building blocks, which collectively form what is called the alphabet:

- We denoted $x, y, z, ...$ for variables. As more variables are necessary sub-indexes will be used, up to countable infinite variables.
- We consider a abstractor connector λ.

Now, we are ready to formally define the untyped λ-calculus:

Definition 3.1.1 (Syntax of Untyped λ-calculus (section 2.1 [22]))**.** A λ-calculus formula (sometimes called term) is defined inductively:

- Every variable $x, y, z...$ is a valid formula.
- If A, B is a formula, then AB is a valid formula.
- If A is a formula and x is a variable, then $\lambda x.M$ is a valid formula.

The set of all variables is denoted by $\mathcal{V}$ and the set of all λ-formulas is denoted by Λ.

Dealing with formal languages, more often than not, we make use of similar inductive statements. Because of this, we find it useful to introduce the Backus-Naur Form notation, noted as BNF [16]. A BNF specification is a set of derivation rules, written as

$$\text{symbols} ::= \text{expression1} \,|\, \text{expression2} \,|...,$$

where symbol is a valid word, and each expression consists of a derived valid formulas. Expressions are separated by the vertical bar: |. For example, we can revisit definition 3.1.1 as follow:

Definition 3.1.2. The formulas of λ-calculus are built via the BNF:

$$A, B ::= x \mid (AB) \mid (\lambda x.A).$$

where x denote any variable.

From this point on we know how the λ-formulas look like. Now it is time to provide meaning. This point is better understood with some examples, using natural numbers. We recommend to trust us in their existence and read this examples first to gain traction in the system. The notion of natural numbers is formalized definition 3.1.16.

Semantics of untyped λ-calculus

Let us now explain the idea behind the formalism. Consider the expression $\lambda x.(x+1)$. This expression represent the idea of the function $f(x) = x+1$ that take a variable x and return $x+1$. $\lambda x.M$ is called the *abstraction* of x.

From the notion of abstraction naturally arises the second one: *application*. Consider formulas $M = \lambda x.x+1$ and $N = 3$. Then $MN = (\lambda x.x+1)(3)$ represent the application of M to N. In untyped λ-calculus, N can be any formula. Thus for example, the formula $\lambda f.\lambda g.fg$ just represent the composition of formulas.

Example 3.1.1. The formula

$$\lambda g.(\lambda f.(\lambda x.(gf)(x)))(x+1)(x+2),$$

can be understood as $(g \circ f)(x)$ where $g(x) = x+1$ and $f(x) = x+2$.

In a expression $M = (\lambda x.N)$ we say that the variable x is *bound* in M.

This notion however has to be formalized. This is done via the notions of α-equivalence, β-equivalence and η-equivalence.

The idea of α-equivalence, $=_\alpha$, is that expressions such as $\lambda x.x$ and $\lambda y.y$ are essentially the same. That is, we consider formulas up to rename of variables. To formalise this, we have to formalize the concept of free and bound variables, and the concept of renamed.

Definition 3.1.3. We have a function $FV : \Lambda \to \mathcal{P}(\mathcal{V})$ defined recursively, such that:

- $FV(x) = \{x\}$ for every $x \in \mathcal{V}$.
- $FV(MN) = FV(M) \cap FV(N)$ for every $M, N \in \Lambda$.
- $FV(\lambda x.M) = FV(M) \backslash \{x\}$ for every $M \in \Lambda, x \in \mathcal{V}$.

We can now define the rename of a variable. We are going to define the more general process of substitution, and define renaming as a particular case. This process is the one behind the intuitive idea of evaluation. For example when we consider $(\lambda y.y^2 + y)(4) = 4^2 + 4 = 20$ we are replacing the value of x by the formula 4.

Definition 3.1.4. The substitution of N for free occurrences of x in M, denoted as $M[N/x]$ is defined recurrently in the structure of λ-formulas by:

$$\begin{aligned}
x[N/x] &\equiv N, && \\
y[N/x] &\equiv y, && \text{if } x \neq y, \\
(MP)[N/x] &\equiv (M[N/x])(P[N/x]), && \\
(\lambda x.M)[N/x] &\equiv \lambda x.M, && \\
(\lambda y.M)[N/x] &\equiv \lambda y.(M[N/x]), && \text{if } x \neq y \text{ and } \notin FV(N), \\
(\lambda y.M)[N/x] &\equiv \lambda y'.((M[y'/y])[N/x]), && \text{if } x \neq y \text{ and } y \in FV(N), \\
& && \text{with } y' \notin FV(N) \cup \{x\}.
\end{aligned}$$

When $N = y$ a variable we say that $[N/x] = [y/x]$ is a rename.

Definition 3.1.5. We define the α-equivalence $=_\alpha$ as the smallest congruence relation on Λ such that:

$$\lambda x.M =_\alpha \lambda y.M[x/y], \qquad \forall y \in \mathcal{V} \backslash FV(M).$$

In other words is the congruence that consider every rename of a bounded variable as equals. This type of property-oriented definition is commonplace in λ-calculus, as it allows for some synthetic and goal-oriented definition. More often than note we will not give explicit use of this equivalence.

Using this same tool we are going to define the β-reduction. This is not going to be an equivalence implication, but rather a relationship. It abstracts the notion of 4 and $(\lambda x.2 + x)(2)$ being equals. Formally:

Definition 3.1.6 (Section 2.5 [22])**.** We define the *single-step β-reduction* as the smallest relationship $\to_\beta$ such that:[1]

$$(\beta) \qquad \frac{}{(\lambda x.M)N \to_\beta M[N/x]},$$

$$(\text{cong}_1) \qquad \frac{M \to_\beta M'}{MN \to_\beta M'N},$$

$$(\text{cong}_2) \qquad \frac{N \to_\beta N'}{MN \to_\beta MN'},$$

$$(\zeta) \qquad \frac{M \to_\beta M'}{(\lambda x.M) \to_\beta \lambda x.M'}.$$

In this definition we can see that rule (β) is the main objective, while the others are sanitary conditions.

Definition 3.1.7. We define the *multiple-step β-reduction* $\twoheadrightarrow_\beta$ as the reflexive, transitive closure of $\to_\beta$.

Definition 3.1.8. We define the β-equivalence $=_\beta$ as the symmetric closure of $\twoheadrightarrow_\beta$.

While most constructions will be understood clearly using only β-reduction, we have to define the concept of η-reduction for a full picture of the system.

Up to this point the focus of the system was to define an intensional language for computation. The η-equivalence provides us with the tools to consider λ-calculus in a extensional way.

Definition 3.1.9. We define the single-step η-reduction $\to_\eta$ as the smallest relationship such that:

$$(\eta) \qquad \frac{}{(\lambda x.Mx) \to_\eta M} \qquad \forall x \notin FV(M),$$

$$(\text{cong}_1) \qquad \frac{M \to_\eta M'}{MN \to_\eta M'N},$$

$$(\text{cong}_2) \qquad \frac{N \to_\eta N'}{MN \to_\eta MN'},$$

$$(\zeta) \qquad \frac{M \to_\eta M'}{(\lambda x.M) \to_\eta \lambda x.M'}.$$

Similarly, we define the multiple-step η-reduction $\twoheadrightarrow_\eta$ as the transitive reflexive closure of $\to_\eta$, and the η-equivalence as the symmetric closure of $\twoheadrightarrow_\eta$.

[1] Should i explain the notation $\frac{A}{B}$

Definition 3.1.10. We define the single-step $\beta\eta$-reduction $\to_{\beta\eta}$ as the union of $\to_\beta$ and $\to_\eta$. We define the multiple-step $\beta\eta$-reduction $\twoheadrightarrow_{\beta\eta}$ as the transitive reflexive closure of $\to_{\beta\eta}$.

3.1.1 Church-Rosser Theorem

This subsection we present an important result for lambda calculus: the *Church-Rosser* theorem. The idea behind this theorem is to prove that every reduction (either β, η or a mix) provide an unified sense of reduction. First, we present some definitions.

Definition 3.1.11. Consider a relation $\to$ and let $\twoheadrightarrow$ be its reflexive transitive closure. We can define three relations:

1. The Church-Rosser Property:

$$M \twoheadrightarrow N, M \twoheadrightarrow P \implies \exists Z : N \twoheadrightarrow Z, P \twoheadrightarrow P.$$

2. The Quasidiamond Property:

$$M \to N, M \to P \implies \exists Z : N \twoheadrightarrow Z, P \twoheadrightarrow P.$$

3. The Diamond Property :

$$M \to N, M \to P \implies \exists Z : N \to Z, P \to P.$$

Remark 3.1.1. Note that, while $3) \implies 1)$, it is not necessary that $2) \implies 1)$.

With this notation, we introduce the Church-Rosser Theorem, proved as done by Martin-Löf. We are going to need some tools. The first one is going to be an alternative definition of $_{\beta\eta}$.

Definition 3.1.12 (parallel one-step reduction)**.** We define the parallel one-step reduction $\triangleright$ as the smallest relationship such that,

$$(1) \qquad \frac{}{x \triangleright x}$$

$$(2) \qquad \frac{M \triangleright M' \qquad N \triangleright N'}{MN \triangleright M'N'}$$

$$(3) \qquad \frac{M \triangleright M'}{(\lambda x.M) \triangleright \lambda x.M'}$$

$$(4) \qquad \frac{M \triangleright M' \qquad N \triangleright N'}{(\lambda x.M)N \triangleright M'[N'/x]}$$

$$(5) \qquad \frac{M \triangleright M'}{(\lambda x.Mx) \triangleright M'} \qquad \forall x \notin FV(M)$$

where x is any variable and M, N, M', N' any formula.

Lemma 3.1.1 (Characterization of $\triangleright$). *Let M, M' be formulas, then:*

1. $M \to_{\beta\eta} M' \implies M \triangleright M'$
2. $M \triangleright M' \implies M \twoheadrightarrow_{\beta\eta} M'$

Proof. 1. Using (1), (2) and (3) in an induction argument derives in $N \triangleright N$ for any formula. Then, we assume that $M \to_{\beta\eta} M'$. We apply induction on the structure of $\to_{\beta\eta}$ and conclude proving that every rule use for $\to_{\beta\eta} M'$ implies $M \triangleright M'$.

- (β) In this case, then $M = (\lambda x.Q)M$ and $M' = Q[N/x]$. Then using (4) $M \triangleright M'$.
- (η) In this case, then $M = (\lambda x.Qx)$ and $M' = Q$. Then using (5) $M \triangleright M'$.
- (cong_1) In this case, then $M = PQ$ and $M' = PQ'$ for some $Q \to_{\beta\eta} Q'$. Using induction $Q \triangleright Q'$ and using (5) $M \triangleright M'$.
- (cong_2) Analogous.
- (ζ) In this case, then $M = \lambda x.Q$ and $M' = \lambda x.Q'$ for some $Q \to_{\beta\eta} Q'$. Using induction $Q \triangleright Q'$ and using (3) $M \triangleright M'$.

Where, in every point, x denote some variable, and M, N, M', N', P, Q, Q' denote some formula.

2. Similarly, every possible in which $M \triangleright M'$ rule derive in $M \twoheadrightarrow_{\beta\eta} M'$:
   (1) By reflexivity of $\twoheadrightarrow_{\beta\eta}$.
   (2) By $\text{cong}_1, \text{cong}_2$ in either definition of $\to_\beta$ and $\to_\eta$.
   (3) By ζ in either definition of $\to_\beta$ and $\to_\eta$.
   (4) Then we have $(\lambda x.M)N \triangleright M'[N'/x]$ with $M \triangleright M' \qquad N \triangleright N'$. By induction $M \twoheadrightarrow_{\beta\eta} M'$ and $N \twoheadrightarrow_{\beta\eta} N'$. By transitive closure:

$$(\lambda x.M)N \twoheadrightarrow_{\beta\eta} (\lambda x.M')N \twoheadrightarrow_{\beta\eta} (\lambda x.M')N' \twoheadrightarrow_{\beta\eta} (M')[N'/x].$$

   (5) Then we have $(\lambda x.Mx) \triangleright M'$, for some $M \triangleright M'$ and for some $x \notin FV(M)$. Finish using (cong_2) and η.

Where, in every point, x denote some variable, and M, N, M', N', P, Q, Q' denote some formula. □

Remark 3.1.2. As a consequence, $\twoheadrightarrow_{\beta\eta}$ is the reflexive, transitive closure of $\triangleright$.

Lemma 3.1.2 (Subtitution Lemma). *If $M \triangleright M'$ and $U \triangleright U'$, then $M[U/y] \triangleright M'[U'/y]$.*

Proof. As we define in 3.1.4 a capture-avoiding substitution, in the sense that we do not let bound variable to be substituted, we can assume without any loss of generality that M. Similarly to previous lemma, we proceed by induction, studying the last rule applied. The induction is proceed on the rules applied to $M \triangleright M'$.

(1) In this case $M = M' = x$. If $x \neq y$ the substitution does not alter M so we have finished. If $x = y$, the formula is only $U \triangleright U'$, that we know by hypothesis.

(2) In this case $M = NP, M' = P'N'$ for some formulas N, N', P, P' such that $N \triangleright N', P \triangleright P'$. Proceed by induction on these implications and apply (2).

(3) In this case $M = \lambda x.N$ and $M = \lambda x.N'$, for some $N \triangleright N'$. Apply induction on N and end by (3).

(4) Then we have $(\lambda x.M)N \triangleright M'[N'/x]$ with $M \triangleright M'$ and $N \triangleright N'$. By induction on both of the relationship, and applying (4).

(5) Then we have $(\lambda x.Mx) \triangleright M'$, for some $M \triangleright M'$ and for some $x \notin FV(M)$. Finish using induction on $M \triangleright M'$ and by (5).

□

While the proof is rather straight forward, one can see that it does require induction, and thus the length of it. [2]

Definition 3.1.13 (Maximal parallel one-step reduction)**.** Given a formula M, the *maximal parallel one-step reduction* $M*$ is defined inductively:

- $(x)^* = x$
- $(PN)* = P^*N^*$ but for β-reductions.
- $((\lambda x.P)N)^* = Q^*[N/x]$
- $(\lambda x.N)^* = \lambda x.N^*$ but for η-reductions,
- $(\lambda x.Nx)^* = N^*$

where x is any variable, and N, P is any formula.

Lemma 3.1.3 (Maximal Parallel one-step)**.** *If $M \triangleright M'$ then $M' \triangleright M^*$.*

Proof. We proceed by induction on M again:

(1) In this case $M = M' = x$. Then $M^* = x$.

(2) In this case $M = NP, M' = P'N'$ for some formulas N, N', P, P' such that $N \triangleright N'$, $P \triangleright P'$.

- If NP is not a β-reduction, then $M^* = P^*N^*$ and we can use the hypothesis induction on N^* and P^* and (2).
- If NP is a β-reduction, then $N = \lambda x.Q$, thus $M^* = Q^*[N^*/x]$. $P = \lambda x.Q$ and $P \triangleright P'$ could be derived using congruence to λ-abstraction (2) or extensionality (5). In the first case use induction on Q and (4). On the second case $Q = Rx$, use induction on R, and apply substitution lemma.

[2] Maybe add some explanation here.

Proceed by induction on these implications and apply (2).

(3) In this case $M = \lambda x.N$ and $M = \lambda x.N'$, for some $N \triangleright N'$. If M is not an η-reduction, then use induction hypothesis and finish with (3). Otherwise we have that $N = Py$, and distinguish two cases on the last rule applied to N:

(2)$\rightarrow$ (3) Then apply induction hypothesis to P and end using (5).

(4)$\rightarrow$ (3) We have that $N = P = \lambda y.Q \triangleright N' = Q'[x/y]$. Apply induction hypothesis using that $M' = \lambda x.N' = \lambda x.Q'[x/y] = \lambda y.Q'$.

(4) Then we have $(\lambda x.P)N \triangleright P'[N'/x]$ with $P \triangleright P'$ and $P \triangleright P'$. By substitution lemma.

(5) Then we have $(\lambda x.Px) \triangleright P'$, for some $P \triangleright P'$ and for some $x \notin FV(P)$. Finish using induction on

□

Remark 3.1.3. We have thus proved that $\triangleright$ have the diamond property.

Remark 3.1.4. If a relation $\rightarrow$ satisfy the diamond property, its closure $\twoheadrightarrow$ satisfy the Church-Rosser property.

As a direct consequence of the previous lemmas, we prove the Church-Rosser theorem.

Theorem 3.1.4. *Church-Rosser $\twoheadrightarrow_{\beta\eta}$ satisfy the Church-Rosser property.*

Now let take a moment to comment on the usefulness of each reduction. We can say that β reduction is the soul of computation while η is useful to cleanup the results. In fact, Church-Rosser is also stated in an alternative form that only includes β-reduction, with a rather similar proof that we omit, but can be checked in [Theorem 1.32, [11]].

Theorem 3.1.5. *$\twoheadrightarrow_{\beta}$ satisfy the Church-Rosser property.*

This theorem gives coherence to the definitions, as it states that any two deduction are basically the same but for the order of the steps. Therefore we can consider λ-calculus and operate, with α-equivalence and $\twoheadrightarrow_{\beta\eta}$ reduction to derives terms. This can be used to define *normal* forms for untyped lambda-terms. More infomation on normalization can be found on [22].

3.1.2 Fixed points and Programming

In this subsection we expose the basic rudiments for programming in untyped lambda calculus. In particular, we explain how to define, for example, a booleans, an integer type and a recursion operator, thus showing the computational potential of recursively enumerable languages. We start with the booleans.

Definition 3.1.14 (Boolean in untyped lambda-calculus)**.** We define the two values T and F, true and false, as:

$$T = \lambda xy.z$$
$$F = \lambda xy.z$$

We also define the operations:

$$\text{Not} = \lambda a.aFT$$
$$\text{And} = \lambda ab.abF$$
$$\text{Or} = \lambda ab.aTb$$

We can check that this construction left us with the basis of a boolean logic, after checking the truth values of the different operations provided. This definition of truth value is really convenient, as it allow us to easily implement control flow of programs.

Definition 3.1.15. We define the $\text{If} = \lambda x.x$.

We can see that in this case, If $TMN = M$ and If $FMN = N$. This construction is very natural and widely used in computer programs. Next, we will define Naturals number.

Definition 3.1.16 (Natural numbers in untyped lambda-calculus)**.** Let f, x be fixed lambda terms, and write $f^n x = f(f(f(...(f(x))...)))$. Then, for each $n \in \mathbb{N}$, we define the *nth Church numeral* as $\overline{n} = \lambda fx.f^n x$.

Finally, we consider the notion of recursive function. This is done with a elegant artefact, based on an idea of fixed points. Let's begin:

Theorem 3.1.6 (Fixed points)**.** *For every term F in untyped lambda calculus, there is a fixed point.*

Proof. Let $A = xy.y(xxy)$ and $\Theta = AA$. Then, for every lambda-term F we have that $N = \theta F = FN$, thus being a fixed point. In fact:

$$N = \Theta F = AAF = (\lambda xy.y(xxy))AF \twoheadrightarrow_\beta F(AAF) = F(\Theta F) = FN.$$

□

The proof of this theorem led to a new definition:

Definition 3.1.17. The term Θ used in the previous theorem is called *Turing fixed point combinator*.

This existence of fixed point theorem is really useful, as we can now define *recursion*. The idea is to define a function as a term with itself as a parameters and proceed to evaluate a fixed point. Let us first present an example.

Definition 3.1.18. We define the terms:

$$\begin{aligned} \text{add} &= \lambda nmfx.nf(mfx), \\ \text{mult} &= \lambda nmf.n(mf), \\ \text{iszero} &= \lambda nxy.n(\lambda z.y)x, \\ \text{predecesor} &= \lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda u.u). \end{aligned}$$

Suppose that we want to define the factorial such that

$$\text{fact}\, n = \text{If}(\text{iszero}\, n)(1)(\text{mult}(n)(\text{fact}(\text{pred}(n))),$$

in orther to do that, we can simply see of all terms, what is a fixed point of:

$$\lambda f.\lambda n.\, \text{If}(\text{iszero}\, n)(1)(\text{mult}(n)(f(\text{pred}(n))),$$

and thus $\text{fact} = \Theta\lambda f.\lambda n.\, \text{If}(\text{iszero}\, n)(1)(\text{mult}(n)(f(\text{pred}(n)))$. In general:

Definition 3.1.19. Given an stop condition g, an stop value s and a recursive step f, we define the recursive term F that computes (g, s, f) as

$$F = \Theta\lambda f.\, \text{If}(gn)(n)(f(\text{pred}(n))$$

Another implementation of the fixed point operator is *Curry's paradoxical fixed point operator*:

Definition 3.1.20. We define Curry's paradoxical fixed point operator Y as:

$$\text{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

This operator is used for the Curry's paradox.

3.1.3 Rosser-Kleene and Curry's Paradoxes

In this subsection we explain the deficits in untyped λ-calculus, that was the original Church construction, that eventually leads into the definition of *typed λ-calculus*.

An aproximation of this argument can be done if we consider domains of functions to be sets. In particular untyped λ-calculus allow functions to be applied to themselves. This is going to be troublesome, considering domains as sets, as this will be a clear infringement of *ZF Axiom Theory*[17]. Namely, the infinite descending sequence

$$f \ni \{f, f(f)\} \ni f \ni \{f, f(f)\}...$$

in contradiction with the *regularity axiom*. An approach that seek solving this problems is presented in the next section, starring the introduction of types.

This last idea, despite being clear and loud to the intuition, is not a proof of inconsistency as there is no need for domains to be sets. Nonetheless, there was another problem that is to be deal that was noticed before. -calculus was proposed to be a deductive system. The *Kleene-Rosser paradox*, first exhibited in [15] proved that simply typed lambda calculus is inconsistent. This paradox was perfected in 1958 by Curry [9] with the so called *Curry's paradox*.

To solve these problems, a very natural idea is included: the use of types in our computation system. This idea is very natural nowadays, as nearly everyone learns to program priors to a deep academic career. However, we must not forget that the origin was not to use types because we liked them initially, but because they really allow us to avoid logical problems.

3.1.4 Lambda calculus as a computation system

Around 1930, people start thinking about what mean for a function to be computable. There were three major answer to this question (section 4, [3]):

- In 1930s Alonzo Church introduces the notion of λ-calculus, in which a function is computable if we can write it as a recursive-lambda term onto Church's numerals, that is, if we can deduce the result after a (hopefully finite) deduction steps.
- Later, Alan Turing, who had previously been a doctoral student of Alonzo Church, developed the concept of the Turing machine, in which a function is computable if a tape machine with a limited set of operations is capable of reproducing it.
- Meanwhile, Gödel define the computable function as the minimum set of function such that some properties (such as the existence of a successor function) are includes, and is closed under some operations.

The goal of this three mathematicians was to solve the Entscheidungsproblem[10].

The interest for this type of problems was clear for the mathematical community as the goal was to propose an algorithm that solve every theorem and can be computed. The Church-Turing thesis fromally proved that these three independently generated notions were in fact equivalent[8].

This thesis is also formulated in a philosophical background, to state that every effective computable function system is equivalent to those three. While it can not be formally proven, it has near-universal acceptance to date. Any computing system that can replicate, and thus is equivalent, to either Turing Machines Calculus or Lambda Calculus is said to be *Turing Complete*.

Nonetheless, after the formalization of the solution of this problem there were a last problem that was not adverted: the finiteness of time. SAT is the problem that has as input a propositional logic formula and solve whether it is

satisfiable. This problem is NP-complete, quite famously the firs of these problems[7]. If we consider the analogous problem for first order algebra, we have an even more complex P-space-complete problem, being almost intractable in worst cases to this date. As a happy consequence for us, much work is left to be done in this area [6].

3.2 Typed Lambda Calculus

Typed λ-calculus is a refinement of untyped λ-calculus, presented as a for the Rosser-Kleene paradox. In this we will introduce the concept of typing. This idea is intended to restrict the ability of types to combine with each other. We start defining *simply typed λ-calculus*, on which we only consider a handfull of basic types, and product and function types. In opposition we can study *dependent typing*, where types can be construct onto each other.

3.2.1 Definition

We synthesize the definitions of typing presented in [18] or [22]. We structure this definition in three steps: types, formulas (sometimes called terms) and equations.

Definition 3.2.1. The types of simply typed λ-calculus are built via the BNF:

$$A, B ::= \iota | \ A \to B \mid A \times B \mid 1.$$

where ι denote a basic type. [3]

Lets start laying some bricks, useful for the intuition. ι type is foundational type. We consider also the type 1. Others author, such as [18] prefer to include a type for every natural. Nonetheless, we consider that we can repeat the construction realized in untyped lambda-calculus to consider this typing.

As it happens in sets, where we have that we can consider the set of functions between two set, here we can consider the type of function between two types. Surely this can get to mind a categorical construction already presented. We continue defining the formulas. Further on, we will use formulas and terms interchangeably. This types are called *function types*.

Definition 3.2.2. The types of simply typed λ-calculus are built via the BNF:

$$A, B ::= \iota \mid A \to B \mid A \times B \mid 1.$$

where ι denote a basic type.

while some authors prefer to include in the definition a type for every natural $\mathbb{N}$, we choose to include only the ι type

[3] Understand the relation with the construction of natural types.

Definition 3.2.3. The *raw typed lambda terms* of simply typed λ-calculus are built via the BNF:

$$A, B ::= x \mid AB \mid \lambda x^t.A \mid \langle A, B\rangle \mid \pi_1 A \mid \pi_2 A \mid *\,.$$

where x denote any variables, and t denote any type.

In this definition we intend to define π_i in a way that $\pi_i\langle A_1, A_2\rangle = A_i$, and such that $*$ is the only term of type 1.

The main difference with untyped calculus, is that every bound variable has a type. That is, for the expression $(\lambda x^t.M)N$ to be coherent we require N to be of type t. This led us to require some sort of condition to N for being of type t.

Nonetheless raw terms have some other meaningless terms, such as projections of a non-pair $\pi_1(\lambda x^t.x)$. We solve all this problems at once with the *Typing rules*. This rules will restricts the semantics of terms. We start by defining *typing context*.

Definition 3.2.4. When a term M is of type t, denoted as $M : t$. A typing context is a set of assumption $\Gamma = (x_1 : A_1)$, on which we assume each variable x_i to be of type A_i.

From this definition, we can state the typing rules, from which we will be able to provide typing. We use the notation $\Gamma \vdash M : A$ meaning that the typing context Γ derives in term M being of type A.

Definition 3.2.5 (Typing Rules)**.** We define the following typing rules, for every typing context Γ.

- Every variable is of the type marked by the context, namely:

$$(var) \qquad \frac{}{\Gamma, x : A \vdash x : A}.$$

- From a term of type $A \to B$ we can derive a term of type B, from a term of type A:

$$(app) \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}.$$

- The projections takes types pairing to each typing:

$$(\pi_i) \qquad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_i M : A_i}, \qquad i = 1, 2.$$

- Pairs takes types to types pairings:

$$(pair) \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \times B}.$$

- $*$ is of type 1.

$$(*) \qquad \frac{}{\Gamma \vdash * : 1}.$$

Note that not every term can be typed, namely the two previous examples of bad-behaviour: $\pi_1(\lambda x^t.x)$ and $(\lambda x^t.M)N$ where N is not of type t. Thus, in simply typed lambda calculus, we only work with typed terms.

Definition 3.2.6. A *simply typed lambda term* is a raw typed lambda term that, together with a typing context of every variable, is subject to be typed.

By assigning type we mean assigning types to the free and bound variables. Further on, when talking about terms we will refer to simply type lambda terms. On these terms, we can define substitution just like we defined it on untyped lambda calculus.

3.2.2 Unification of typing

We can consider two standards in typing: *Church Style* and *Curry Style*.

- Church style, first shown in [4], is an explicit system, as we do only consider expression with well defined types, in the same way that a function is consider with regard to it domain and codomain. In summary, the typing of a function is the first part of the definition of it, so you can define $f : (0,1) \to (0,1)$ and just then define $f(x) = 1/x$. This domain is not it only possible domain, but it what we choose it to be.
- Curry style on the other hand, gain some of the notions of untyped λ-calculus and consider expression as function. To continue the real function analogy, it consider that the function $f(x) = x^2$ just exists, and one can check that it is well defined on $\mathbb{R}$.

More information about the typing style can be found on Chapter 10 and 11 of [11].

Despite this apparent differences, both typing styles can be *unified*. An unifier is a pair of substitution that makes two type styles equal as typed templates. Such an unifier is based onta an algorithm based on type inference. This is instrumental to languages as relevant `python`, that works with implicit typed variables. How this algorithms for unification is shown in great detail in chapter 9 of [22]. Due to this unification, we can consider only explicitely typed terms without any remorse.

3.2.3 Church-Rosser on simply typed lambda calculus

To talk Church-Rosser, we have to talk reduction. Reductions are basically the same but for the additional structure added that we need to respect:

Definition 3.2.7 (Section 2.5 [22])**.** We define the *single-step β-reduction* as the smallest relationship $\to_\beta$ such that:

(β): $(\lambda x^t.M)N \to_\beta M[N/x]$.

$(\beta_{\times,i})$: $\pi_i\langle M_1, M_2\rangle \to_\beta M_i$.

(cong_1): If $M \to_\beta M'$ then $MN \to_\beta M'N$.

(cong_2): If $N \to_\beta N'$ then $MN \to_\beta MN'$.

(ζ): If $M \to_\beta M'$ then $(\lambda x.M) \to_\beta \lambda x.M'$.

We define the *multiple-step β-reduction* $\twoheadrightarrow_\beta$ as the reflexive, transitive closure of $\to_\beta$.

Definition 3.2.8. We define the single-step η-reduction $\to_\eta$ as the smallest relationship such that:

(η): $(\lambda x.Mx) \to_\eta M$,para todo $x \notin FV(M)$.

(η_1): $\langle \pi_1 M, \pi_2 M\rangle \to_\eta M$.

$(\eta_\times)$: If $M : 1$ then $M \to_\eta *$.

(cong_1): If $M \to_\eta M'$ then $MN \to_\eta M'N$.

(cong_2): If $N \to_\eta N'$ then $MN \to_\eta MN'$.

(ζ): If $M \to_\eta M'$ then $(\lambda x.M) \to_\eta \lambda x.M'$.

Similarly, we define the multiple-step η-reduction $\twoheadrightarrow_\eta$ as the transitive reflexive closure of $\to_\eta$, and the η-equivalence as the symmetric closure of $\twoheadrightarrow_\eta$.

We can now talk about about Church-Rosser. Lets check that it does not hold. For example, if $x : A \times 1$ then we can consider $M = \langle \pi_1 x, \pi_2 x\rangle$. Because of the rule η_1, we have that $M \to_\eta \langle \pi_1 x, *\rangle$, but taking $\eta_\times$ into account we can check that $M \to_\eta x$.

Despite this, β-reduction does maintain the Church-Rosser property, so from a computational point of view there is no problem at all.

Chapter 4

Curry-Howard-Lambeck bijection

4.1 Property oriented Propositional Logic

4.2 Curry-Howard bijection

4.3 Lambeck bijection

Bibliography

[1] Rod Adams. *An Early History of Recursive Functions and Computability: From Gödel to Turing*. Docent Press, 2011.

[2] Stefan Banach. "Théorie des opérations linéaires". In: (1932).

[3] Felice Cardone and J Roger Hindley. "History of lambda-calculus and combinatory logic". In: *Handbook of the History of Logic* 5 (2006), pp. 723–817.

[4] Alonzo Church. "A formulation of the simple theory of types". In: *The journal of symbolic logic* 5.2 (1940), pp. 56–68.

[5] Alonzo Church. "A set of postulates for the foundation of logic". In: *Annals of mathematics* (1932), pp. 346–366.

[6] Stephen Cook. "The P versus NP problem". In: *The millennium prize problems* (2006), pp. 87–104.

[7] Stephen A Cook. "The complexity of theorem-proving procedures". In: (1971), pp. 151–158.

[8] B Jack Copeland. "The church-turing thesis". In: (1997).

[9] Haskell Brooks Curry et al. *Combinatory logic*. Vol. 1. North-Holland Amsterdam, 1958.

[10] David Hilbert and Wilhelm Ackermann. *Principles of mathematical logic*. Vol. 69. American Mathematical Soc., 1999.

[11] J Roger Hindley and Jonathan P Seldin. *Lambda-calculus and Combinators, an Introduction*. Vol. 13. Cambridge University Press Cambridge, 2008.

[12] Thomas Hofmeister et al. "A Probabilistic 3—SAT Algorithm Further Improved". In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer. 2002, pp. 192–202.

[13] William A Howard. "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[14] Daniel M Kan. "Adjoint functors". In: *Transactions of the American Mathematical Society* 87.2 (1958), pp. 294–329.

[15] Stephen C Kleene and J Barkley Rosser. "The inconsistency of certain formal logics". In: *Annals of Mathematics* (1935), pp. 630–636.

[16] Donald E Knuth. "Backus normal form vs. backus naur form". In: *Communications of the ACM* 7.12 (1964), pp. 735–736.

[17] Kenneth Kunen. *Set theory an introduction to independence proofs*. Elsevier, 2014.

[18] Joachim Lambek and Philip J Scott. *Introduction to higher-order categorical logic*. Vol. 7. Cambridge University Press, 1988.

[19] Saunders Mac Lane. *Categories for the working mathematician*. Vol. 5. Springer Science & Business Media, 2013.

[20] Bartosz Milewski. *Category theory for programmers*. Blurb, 2018.

[21] Emily Riehl. *Category theory in context*. Courier Dover Publications, 2017.

[22] Peter Selinger. "Lecture notes on the lambda calculus". In: *arXiv preprint arXiv:0804.3434* (2008).

[23] Sigur. *Using Tikz-cd to Illustrate Composition of Natural Transformations*. Computer Science Stack Overflow. URL: https://tex.stackexchange.com/questions/505322/using-tikz-cd-to-path-to-illustrate-composition-of-natural-transformations%20(version:%202021-02-05).

[24] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem. A correction". In: *Proceedings of the London Mathematical Society* 2.1 (1938), pp. 544–546.

[25] Kinoshita Yoshiki. "Prof. Nobuo Yoneda passed away". In: (1996). URL: https://www.mta.ca/~cat-dist/catlist/1999/yoneda.